

C++

1108 서민준

# 함수 오버로딩 (1)

다른 매개변수를 가진 같은 이름인 여러 함수를 만들 수 있는 C++의 기능

```
int MyFunc(int num)
{
    return num + 1;
}
```

```
int MyFunc(int num1, int num2)
{
    return num1 + num2;
}
```

호출 형태를 보고  
어떤 함수를 호출하는 건지  
알 수 있지 않을까?

# 함수 오버로딩 (2)

C++ 컴파일러는 이름이 같은 함수가 정의되었을 때 다음 조건 중 하나라도 만족하면 오버로딩을 허용한다.

## 1. 매개변수의 자료형이 다르다!

```
int MyFunc(int num)
{
    return num + 1;
}
```

```
int MyFunc(double num)
{
    return num + 1;
}
```

## 2. 매개변수의 수가 다르다!

```
int MyFunc(int num)
{
    return num + 1;
}
```

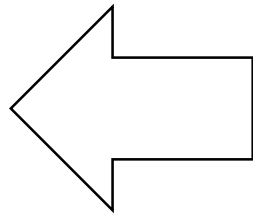
```
int MyFunc(int num1, int num2)
{
    return num1 + num2;
}
```

# 함수 오버로딩 (3)

함수의 반환형은 함수 오버로딩에 고려되지 않는다.

```
int MyFunc(int num)
{
    return num + 1;
}
```

```
double MyFunc(int num)
{
    return num + 1;
}
```



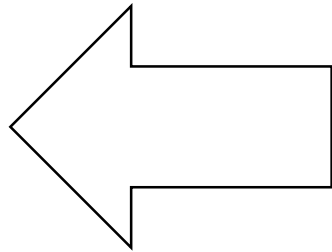
위에서 정의한 MyFunc의 재정의로  
판단하고 **컴파일 오류**를 일으킨다.

# 매개변수의 디폴트 값 (1)

매개변수에 값을 전달하지 않았을 때 매개변수에 특정 값을 전달하고 싶을 때 사용하는 기능

```
int MyFunc(int num = 0)
{
    return num + 1;
}
```

```
int main()
{
    MyFunc();
    return 0;
}
```



이런 식으로 매개변수의 값을 전달하지 않았을 때는 자동으로 디폴트 값이 매개변수에 대입된다.

# 매개변수의 디폴트 값 (2)

디폴트 값은 **여러 개** 가질 수 있다.

```
int MyFunc(int num1 = 0, int num2 = 0)
{
    return num1 + num2;
}
```

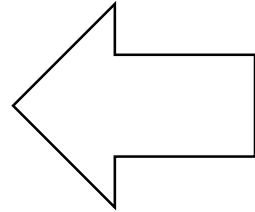
위 함수는 다음과 같은 형태로 호출될 수 있다.

```
MyFunc(); // num1, num2에 모두 디폴트 값 0이 전달됨.
MyFunc(1); // num1에는 1이, num2에는 디폴트 값 0이 전달됨.
MyFunc(2, 3); // num1에는 2가, num2에는 3이 전달됨.
```

# 매개변수의 디폴트 값 (3)

디폴트 값은 **오른쪽**에서부터 채워야 한다.

```
int MyFunc(int num1 = 0, int num2)
{
    return num1 + num2;
}
```

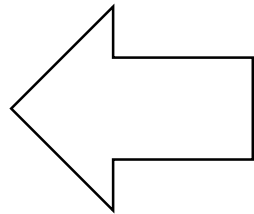


다음과 같은 정의는 컴파일 오류

선언된 후에는 디폴트 값을 다시 선언할 수 없다.

```
int MyFunc(int num = 0);
```

```
int MyFunc(int num = 0)
{
    return num + 1;
}
```

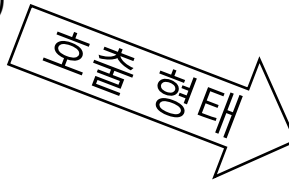


다음과 같은 정의는 컴파일 오류

# 매개변수 디폴트 값 (4)

함수 오버로딩이 가능하나 주의해야 한다.

```
int MyFunc(int num1 = 0, int num2 = 0)
{
    return num1 + num2;
}
```



```
int MyFunc(int num)
{
    return num + 1;
}
```

MyFunc(5, 6);  
MyFunc(4);  
MyFunc();

```
int MyFunc()
{
    return 0;
}
```

컴파일러가 어떠한 함수를 호출한 것인지 구분할 수 없다.



# 매크로 함수

## 특징

전처리가 처리한다.

## 장점

- 일반적인 함수의 비해 **실행속도가 빠르다.**

## 단점

- **정의하기 어렵다.**
- 정의에 **한계가 있다.**
- **디버그가 어렵다.**

```
#define SQUARE(x) ((x) * (x))
```

# inline 함수

```
inline int SQUARE(int x)
{
    return x * x;
}
```

## 장점

- 실행속도가 일반함수보다 빠르다.
- 정의하기 쉽다.
- 디버그가 쉽다.

## 단점

- 자료형에 의존적이다.

## 특징

컴파일러가 처리한다.

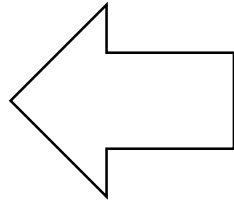
함수가 인라인화에 해가 된다고 판단하면 inline 키워드를 무시

반대로 일반 함수도 인라인화했을 때 성능에 득이 된다고 판단하면 인라인화를 진행하기도 함

# namespace (1)

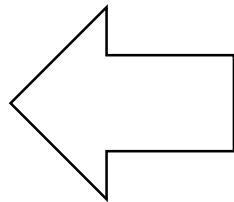
```
int data;
```

```
int MyFunc(int num)
{
    return num + 1;
}
```



기본적으로 전역 namespace에 정의된다.

```
int MyFunc(int num)
{
    return num + 1;
}
```



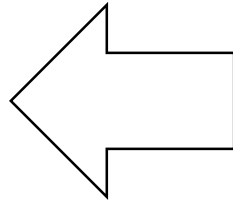
두 함수가 모두 전역 namespace에 정의되어 있다면  
재정의 문제로 컴파일 오류가 발생한다.

```
int MyFunc(int num)
{
    return num + 1;
}
```

# namespace (2)

**namespace** Ex1

```
{  
    int MyFunc(int num)  
    {  
        return num + 1;  
    }  
}
```



다음과 같이 정의하면 정의된 공간이 서로 다른 것으로 인식하여 아무런 문제를 발생시키지 않는다.

**namespace** Ex2

```
{  
    int MyFunc(int num)  
    {  
        return num + 1;  
    }  
}
```

# namespace (3)

**namespace** Ex1

```
{  
    int data;  
  
    int MyFunc(int num)  
    {  
        return num + 1;  
    }  
}
```

namespace 안에 정의된 변수나 함수를 사용하는 방법은 2가지가 있다.

## 1. 범위 지정 연산자( :: )를 사용한다.

```
Ex1::data = 1;  
Ex1::MyFunc(3);
```

## 2. using 명령문을 사용한다.

```
using namespace Ex1;
```

```
data = 2;  
MyFunc(4);
```

# namespace (4)

namespace 내에 있는 함수의 선언과 정의는 다음과 같이 분리한다.

```
namespace Ex1
{
    int MyFunc(int num);
}

int Ex1::MyFunc(int num)
{
    return num + 1;
}
```

namespace는 다른 namespace에 중첩될 수 있다.

```
namespace Ex1
{
    namespace Ex2
    {
        int MyFunc(int num)
        {
            return num + 1;
        }
    }
}
```

# bool형

참과 거짓의 표현을 위한 키워드 **true**, **false**를 저장할 수 있는 자료형

```
bool isTrue = true;  
bool isFalse = false;
```

**true**와 **false**는 각각 1과 0을 의미하지 않는다!

'참'과 '거짓' 을 나타내기 위한 1바이트의 크기의 데이터일 뿐입니다.  
숫자와 연동시키지 말고 그냥 '참'과 '거짓' 을 나타내는 논리형으로 생각해주기 바란다.

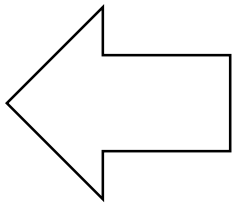
# 참조자 (1)

메모리 공간에 부여된 이름 이외에 새로운 별명을 붙여 참조하는 기능

사용 방법은 자료형& 별명 = 기존 변수명; 의 형식으로 사용된다.

```
int data = 1;  
int& ref = data;
```

```
data = 2;  
ref = 2;
```



이 두 문장 모두 같은 결과를 보인다.



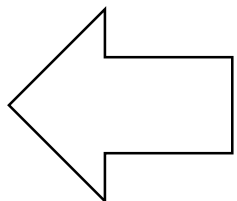
# 참조자 (2)

참조자 사용 방법에는 다음과 같은 특징이 있다.

## 1. 선언과 동시에 초기화해야 한다.

`int& ref;`  다음 문장은 컴파일 에러를 일으킨다.

## 2. NULL값으로 초기화할 수 없다.

`int& ref = NULL;`  다음 문장은 컴파일 에러를 일으킨다.

# 참조자 (3)

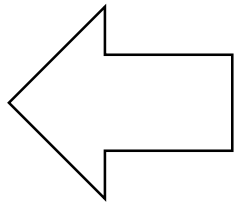
3. 상수 값은 일반적으로 참조할 수 없다.

```
const int num = 0;
```

```
int& ref = num;
```

```
int& ref = 8;
```

```
int& ref = "C++";
```



다음 문장들은 모두 컴파일 오류를 일으킨다.

3-1. 단, **const** 참조자는 참조할 수 있다.

```
const int& ref = "C++";
```

# 참조자 (4)

4. 초기화 후에는 다른 변수를 참조하도록 변경할 수 없다.

```
int data1 = 1;  
int& ref = data1;
```

```
int data2 = 4;  
ref = data2;
```